DESIGN AND ANALYSIS OF AN INSTRUMENTING PROFILER FOR WEBASSEMBLY

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Chandler Gifford

June 2019

© 2019
Chandler Gifford
ALL RIGHTS RESERVED

COMMITTEE MEMBERSHIP

TITLE: Design and Analysis of an Instrumenting Profiler for WebAssembly

AUTHOR: Chandler Gifford

DATE SUBMITTED: June 2019

COMMITTEE CHAIR: Aaron Keen, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: John Clements, Ph.D
Professor of Computer Science

COMMITTEE MEMBER: John Bellardo, Ph.D
Professor of Computer Science

ABSTRACT

Design and Analysis of an Instrumenting Profiler for WebAssembly

Chandler Gifford

This thesis presents the design, implementation, and analysis of WasmProf, an instrumenting profiler for WebAssembly programs. WebAssembly is a compiled language designed for use on the web that, at the time of this writing, is still being actively developed. At present, performance analysis for WebAssembly programs mostly consists of browsers' built-in sampling profilers. These profilers work well in many cases but only give a statistical estimation of the distribution of function calls and are, therefore, not well-suited for more fine-grained analysis. The WasmProf instrumenting profiler fills this analysis gap. WasmProf is capable of tracking the number of calls made and the time spent in every function called within the profiled program. Analysis of WasmProf demonstrates performance equivalent to or slightly better than similar tools that perform instrumentation and dynamic analysis on WebAssembly programs.

ACKNOWLEDGMENTS

I would like to thank the following people who helped me during my time at Cal Poly and made possible the completion of this thesis:

- My parents for their support putting me through school and encouraging me to pursue my MS degree.
- My advisor, Dr. Aaron Keen, for helping me complete this thesis.
- My fellow Cal Poly classmates for creating a great environment and offering support during my time at Cal Poly.

TABLE OF CONTENTS

Page

LIST OF TABLES ix

LIST OF FIGURES x

CHAPTER

1 Introduction 1

2 Background 3

2.1 Dynamic Analysis 3

2.2 Profiling 3

2.2.1 Sampling 4

2.2.2 Instrumenting 4

2.3 WebAssembly and JavaScript 5

2.3.1 WebAssembly Module 5

2.3.2 JIT Compiler 6

2.4 Existing Tools 7

3 Related Work 9

3.1 WebAssembly 9

3.2 Code Profilers 9

3.3 Taint Assembly 10

3.4 Wasabi 10

4 Requirements and Validation 11

4.1 Concrete Requirements 11

4.2 Functional Requirements 12

5 Implementation 13

5.1 Major Design Decisions 13
5.1.1 Instrumentation 13
5.1.2 Information Collected 13
5.1.3 Call Arcs 14
5.1.4 Host API 15
5.2 Limitations of WebAssembly 16
5.2.1 Memory Limitation 16
5.2.2 Access to System Resources 17
5.3 Instrumentation Implementation 17
5.3.1 Call Hoisting 18
5.3.2 WebAssembly Support Functions 19
5.3.3 Decorators 20
5.4 Host Implementation 21
5.4.1 API JavaScript Implementation 21
5.4.2 Runtime Patching 22
5.4.3 Data Structures 23
5.4.4 Printing Results 23
5.5 Special Call Situations 25
5.5.1 Calls to Exported Functions 25
5.5.2 Indirect Calls 27
6 Results 29
6.1 Experimental Setup 29
6.1.1 Profiler Comparisons 29
6.1.2 Benchmark Programs 30
6.1.3 Test Machine 31

6.2 Validation Results 31
6.2.1 Concrete Requirements 31
6.2.2 Functional Requirements 32
6.3 Performance Results 33
6.3.1 Binary Size Difference 33
6.3.2 Compilation Difference 34
6.3.3 Runtime Difference 35
6.4 Usability and Output 38
6.4.1 Instrumentation Usage 39
6.4.2 Runtime Usage 39
6.4.3 Output Inaccuracies 40
7 Future Work 42
7.1 Profiler Improvements 42
7.2 Profiler with Future WebAssembly Features 43
7.3 Other Host Environments 44
8 Conclusion 45
BIBLIOGRAPHY 47
APPENDICES
A Example of Instrumented WebAssembly Importing Required Functions 50
B Program to validate relative function timing of WasmProf 51
B.1 Validation Code 51
B.2 WasmProf Validation Output 53
C Wasabi Profiler 57

LIST OF TABLES

Table		Page
6.1	Binary size of original program and instrumented programs (in kilobytes)	33
6.2	Compilation time of WebAssembly program (in milliseconds)	34
6.3	Runtime of benchmark programs. Normalized to the runtime of the original WebAssembly binary in Firefox.	35

LIST OF FIGURES

Figure		Page
5.1	Example nested call before hoisting and instrumentation	19
5.2	Example nested call after hoisting and instrumentation	19
5.3	Flat profile of a main function that makes one call each to add and double functions.	24
5.4	Call graph profile that shows main calling add and double functions.	25
5.5	Example JavaScript code that calls a WebAssembly function	26
5.6	Example WebAssembly code that imports and calls a JavaScript function	27
6.1	WasmBoy performance comparison of WasmProf, Wasabi, and unmodified program	37
6.2	WasmBoy performance comparison of sampling profiler and unmodified program	38
6.3	Example of inaccurate timing output.	40

Chapter 1

INTRODUCTION

This thesis presents the design, implementation, and analysis of WasmProf, an instrumenting profiler for WebAssembly code. WebAssembly is quickly growing as the target language of choice for deploying high-performance applications to the web. Everything from PDF tools [15] to game engines [16] are using WebAssembly to create programs that were previously too demanding to run reliably on the web. A major focus of WebAssembly is faster performance which means good tools for analyzing and tuning the execution of WebAssembly programs are vital to the development of more applications. Profilers, in particular, are an important performance analysis tool that allows developers to see where their application is spending the most time and consequently, which areas they should focus on tuning. WebAssembly is a relatively new technology that is still being developed and currently few tools exist for the analysis of WebAssembly code. WasmProf is this thesis's contribution to the set of available tools that will help developers tune their WebAssembly code and enable more and more applications to run on the web.

Chapter 2 and Chapter 3 discuss some background on the current state of WebAssembly, as well as some related works in WebAssembly dynamic analysis. We discuss how the WasmProf tool fits into the field of WebAssembly analysis which basically consists of profilers built into most modern browsers as well as research into doing dynamic analysis on WebAssembly code. WasmProf is presented as a tool with a specific purpose; profile function call times and call counts. Specializing the tool provides users with an easy to use profiler that gives in-depth information about function call times while reducing performance overhead on instrumented code as much as possible.

WasmProf aims to produce accurate timing of functions, not necessarily absolute

timings due to the inherent overhead of instrumentation, but function timings that are accurate relative to each other. This provides the user with insight into which functions may require attention and optimization. Chapter 4 details these requirements for WasmProf and how we attempted to validate that WasmProf is behaving as a profiler should; it should correctly track all calls in a program and provide useful function timing.

With WasmProf's goals in mind, the actual design and implementation of the WasmProf tool is discussed in Chapter 5. The chapter explains how design decisions were inspired by previous profilers in other languages and how the WasmProf feature set was restricted by the current WebAssembly specification. Also presented is an API developed to describe how WasmProf should interact with the host environment. The specific implementation presented in this thesis uses JavaScript as the host environment from which WebAssembly programs are instantiated, but this is not necessarily a requirement. The WasmProf profiler implementation was tested on a variety of different WebAssembly programs and Chapter 6 presents the results of those tests. The correct functionality of WasmProf was validated and its performance was tested against similar tools. Results show that WasmProf meets its goals for a working profiler but definitely suffers from some expected downsides due to the nature of instrumentation's high overhead.

Due to the active development of WebAssembly, the landscape of available tools for analyzing WebAssembly code will likely change in the near future. WasmProf meets the goal of implementing an instrumenting profiler, but there are still many additions to be made and different designs to consider that are out of the scope of this thesis. Chapter 7 provides a look at possible future developments in the field of WebAssembly profilers. We present both possible improvements to WasmProf as well as some possible different fundamental profiler designs that could be developed as WebAssembly matures.

Chapter 2

BACKGROUND

This chapter presents background information on performance analysis, code profiling, and WebAssembly. These concepts will be referenced frequently throughout the remainder of this thesis.

2.1 Dynamic Analysis

Dynamic analysis refers to the practice of monitoring a program while it is running and analyzing it in some way. This can take many forms; a dynamic analysis tool could monitor memory usage, trace what code is being executed, measure how long a program takes to run, etc.

For most popular languages there is some sort of dynamic analysis tools for analyzing the performance of a program at runtime. These tools can generally be split into two categories: lightweight sampling analysis which interrupts execution at defined intervals and has relatively low overhead, and heavyweight instrumenting analysis which modifies a program in some way to track its execution. One of the best known and most developed heavyweight dynamic analysis tools is Valgrind for analyzing execution of C programs [12].

2.2 Profiling

Profiling is a special case of dynamic analysis that seeks to measure how long various parts of a program take to run. Profilers can likewise be split into two main categories: sampling profilers and instrumenting profilers.

2.2.1 Sampling

Sampling profilers will generally use a timer to interrupt a program at defined intervals. Each time the program is interrupted, the call stack is examined to determine which function the program is currently in. By keeping track of which functions are hit over the course of many interruptions, the sampling profiler can build a statistical estimation of how much time is spent in each function or section of code. These types of profilers have the advantage that they can be tuned with different interruption intervals in order to adjust how much runtime overhead they add to the original program. The trade-off is the longer the interval between interruptions, the more likely the profiler is to miss function calls and build an inaccurate representation of program behavior. In addition, sampling profilers cannot accurately count exactly how many times each function is called.

2.2.2 Instrumenting

The foil to sampling profilers are instrumenting profilers. These profilers modify the original code either at runtime or prior to being run. They add routines which allow every single event of interest to be tracked so that nothing is missed. The most common example would be tracking the entry and exit time for every single function call. This is much more accurate in terms of building a call graph of what a program is doing but it has a downside. The instrumentation adds a lot more overhead than a sampling profiler which causes the program to take longer to execute and can skew timing results. Generally, an instrumenting profiler can tell which functions are taking more time and count exactly the number of calls, but the absolute timing will likely be higher than the original program.

Both types of profilers have a place in performance analysis and can be useful depending on what information the programmer wants to obtain about their program.

As discussed later in this thesis, many sampling profilers already exist in web browsers; this thesis focuses on the implementation of WasmProf as an instrumenting profiler.

2.3 WebAssembly and JavaScript

WebAssembly is a language (more specifically a virtual instruction set architecture) that is designed to be a compilation target for higher level languages (though it can also be written by hand). WebAssembly has both a binary and a textual encoding. The binary encoding is the main representation and allows for code to be represented more compactly and compiled more efficiently. The textual representation is helpful for debugging or writing code by hand. The WebAssembly architecture is designed as a stack-based virtual machine, meaning each instruction takes its parameters off of the stack and pushes its results back onto the stack. Though this is conceptually how WebAssembly works, specific implementations do not necessarily need to use stack-based execution; it just needs to match the program flow of a stack machine in every way.

2.3.1 WebAssembly Module

All WebAssembly code is organized into modules. Each WebAssembly module consists of the following components (this is not a complete list of everything in WebAssembly but encompasses what is relevant for this thesis):

- linear memory: the linear memory is sandboxed so that WebAssembly code cannot access any outside data unless it is specifically provided (see imports and exports). There is exactly one linear memory for each WebAssembly module; this may be changed in a future version, but at the time of writing WebAssembly 1.0 enforces this rule [14].
- globals: Each module also has a list of globals, which are typed places to store global data. Globals are separate from linear memory which, as seen in Chapter

5, is an important property for WasmProf since it can add globals without worrying about affecting the original program.

- functions: The main code, functions are defined with a type and a body that is a list of instructions that correctly adhere to the WebAssembly format.
- function table: A function table contains references to functions and can be indexed into with the call_indirect instruction. This is how WebAssembly achieves dynamic function pointers. As with linear memory, WebAssembly 1.0 enforces that there is exactly one function table per module.
- imports and exports: Memory, globals, functions, and function tables can also be imported or exported. An import is a declaration of an object (of one of the stated types) that will be implemented in the host environment. An export exposes an object in WebAssembly so it can be used by the host environment. Note that the rule about memory and the function table being singular still applies. (i.e., a WebAssembly module cannot both have a memory declared internally and also import a memory from the host environment)
- custom sections: Finally, WebAssembly allows for custom sections in the code which are not part of the specification but can be used for non-standard implementations of WebAssembly. There is one custom section that is a pseudo-standard and that is the Names section, which provides a list of strings that correspond to the names of the WebAssembly functions. The Names section is extremely useful to WasmProf because it allows the profiler to output meaningful names.

2.3.2 JIT Compiler

An important property of WebAssembly and JavaScript is that they most often are executed using a just-in-time compiler. This compiler reads in the code and compiles it into actual machine code that can be executed. In the case of JavaScript, the

code is first parsed from its textual representation into an internal representation that is used by the compiler. Once the code is parsed, the JIT compiler tries to generate machine code very quickly to get the program running, and then it watches program execution to determine where it can go back and further compile the code with optimization [4]. On the other hand, before execution WebAssembly is already compiled into its binary format which has two main advantages. First, the parsing step is not necessary since the binary file can be read directly into a form used by the JIT compiler. Second, WebAssembly can have optimization run beforehand by the compiler that is compiling a higher level language into WebAssembly so the JIT compiler is immediately compiling to optimized machine code and has to do very little work in monitoring the program and re-optimizing later [9, 5]. This is why WebAssembly is generally considered to be faster than JavaScript. In the case of both languages, the JIT compiler can generate very fast machine code, but in the case of WebAssembly it is much easier and faster to generate this optimized machine code.

2.4 Existing Tools

Since WebAssembly is still in its infancy, development of new tools for the language is fast-paced and changing almost every month. At present, relatively few tools exist for performance analysis of WebAssembly. Existing tools include profilers built into modern JavaScript/WebAssembly engines such as the one in V8, which powers Chrome and Node.js, and SpiderMonkey, which powers Firefox [1, 13]. Both engines can profile WebAssembly code alongside JavaScript to some degree. However, both profilers are sampling profilers that provide a statistical distribution of time spent in function calls. While sampling profilers are useful in most cases as a first pass performance analysis, in many applications there may be a need for a more heavy-handed and precise analysis of code performance.

Additionally, there exist some tools that have been built to do analysis of WebAssembly code using instrumentation. These tools are discussed further in Chapter 3.

Chapter 3

RELATED WORK

3.1 WebAssembly

In [8], Hass et al. present an overview of WebAssembly and its design goals while they were creating the specification. They sought to make WebAssembly safe, fast, portable, and compact. It is safe in that it runs in a secure execution environment similar to JavaScript. It is fast because it can be optimized ahead of time and compiled quickly to machine code on the host. It is portable in that the format does not restrict itself to any one browser or type of hardware. And it is compact because the binary stack machine representation of the code is much more space efficient than JavaScript's textual representation.

3.2 Code Profilers

Profilers, and in particular call graph execution profilers, exist for most popular programming languages. One of the most notable is gprof, a call graph execution profiler for C, Fortran and Pascal programs [7]. Gprof does binary instrumentation during compilation and uses calls to monitoring routines at the start and end of each function that is to be tracked. To reduce overhead, gprof gathers profiling data in memory and then outputs that information to a file as the program exits.

There has also been a lot of past work in developing profilers for the Java Virtual Machine [11], including work to implement sampling profilers [20] as well as the development of more sophisticated instrumentation that can profile all classes in the JDK [3].

Code profilers for profiling JavaScript and WebAssembly also exist in all major browsers [1, 13]. These profilers are sampling profilers, meaning they do not track

the exact execution; they interrupt execution periodically and check which function is being executed. This is used to build a statistical estimation for how much time is spent in each routine.

3.3 Taint Assembly

One existing dynamic analysis tool for WebAssembly is TaintAssembly by Fu et al. TaintAssembly is a data flow tracking modification to the V8 engine that allows for analysis of information flow through a program [6]. As a modification to the V8 engine, TaintAssembly suffers from lack of portability. In addition, it executes code in an interpreted environment which is fine for taint analysis but ill-suited for profiling the performance of WebAssembly programs since performance relies heavily on the code being just-in-time compiled.

3.4 Wasabi

In their recent paper, Lehmann and Pradel sought to create a dynamic analysis tool for WebAssembly called Wasabi [10]. Wasabi is a flexible dynamic analysis tool that performs binary instrumentation of any type of instruction. It takes already compiled WebAssembly binaries and outputs a modified WebAssembly binary as well as supporting JavaScript code that must be loaded alongside the WebAssembly program. Analysis is done using callbacks to JavaScript routines. While this is flexible and easier to use, it has higher overhead than solutions such as gprof due to the fact that cross language calls are slower than tracking profiling metadata in memory.

Wasabi is used as the main point of comparison for the performance of the Wasm-Prof profiler because it allows for the same sort of profiling.

Chapter 4

REQUIREMENTS AND VALIDATION

Validation for this work consists of meeting a number of concrete requirements as well as a number of functional requirements that are more difficult to test and validate quantitatively.

4.1 Concrete Requirements

The first and most important requirement is that an instrumented program produces the exact same result as its uninstrumented version. WasmProf instruments WebAssembly binaries in such a way that it never changes the semantics of the original program (Section 5.3), so by design, this requirement should always hold true. In addition, we validated that the output of programs remains unchanged before and after instrumentation. All of the benchmark programs are simply run with and without instrumentation and their outputs are compared to ensure correctness of the instrumented program.

The second requirement is to make sure the instrumentation method can correctly track entry to and exit from each function. This functionality only represents a small part of the intended profiler functionality, but it is an important step to building a full-fledged profiler and it is much easier to test and validate than the fully-featured profiler. Testing for this requirement is done by generating a program that calls a number of test functions a known number of times. The number of function entries and exits from the instrumented code is compared to the known value for number of function calls to validate that it is working correctly.

4.2 Functional Requirements

The first functional requirement is to keep the overhead from instrumentation low. There is no exact metric for what constitutes low overhead so this requirement is tested by comparison. First, the total runtime of instrumented code is compared to uninstrumented code. Second, the runtime of instrumented code is compared to the runtime in other profilers; both the Wasabi dynamic analysis framework as well as the sampling profilers in Chrome and Firefox are used for comparison. These comparisons are done across a range of benchmarks to make a qualitative determination of how low the overhead of the instrumented code is.

The second functional requirement is that the function-level timings be as accurate as possible. This is difficult to test because there is no perfect reference to compare against; every profiler has some amount of overhead so it is impossible to get timing exact. However, this requirement can be validated by looking at the relative timing results of well-known functions. For example, if a function with 1000 additions is profiled and a function with 9000 additions is profiled, we expect for the latter to take 9 times longer than the former. Also, if the two functions are profiled back-to-back in the same application, we expect the former to make up approximately 10% of the total runtime and the later to make up 90% of the total runtime. Due to the overhead of instrumentation, we do not expect function timings to match how quickly the original program runs. What we want is for an accurate representation of how the timing of different functions in a program compare to each other.

Chapter 5

IMPLEMENTATION

5.1 Major Design Decisions

WasmProf was designed to be as independent as possible from the runtime environment in which it is running. This is enabled by an API (specified in Section 5.1.4) that must be implemented by the host environment, but other than that WasmProf should work in a browser or any other WebAssembly runtime environment.

5.1.1 Instrumentation

The WasmProf profiler is implemented using only instrumentation to determine function run times and counts. Since the goal is to profile WebAssembly code without modifying the runtime engine, there is no access to any sort of preemption that would allow for sampling. In any case, sampling profilers are already built into many browsers.

WasmProf tries to track as much information about the program completely in WebAssembly code. Data is stored in globals and only sent out to the host on exit from a top-level WebAssembly function. Sending out data to the host is done via the host API functions defined in Section 5.1.4.

5.1.2 Information Collected

WasmProf stores two main data points for function performance. The number of times a function is called, and how much cumulative time is spent in each function. From this data various metrics about function runtime can be calculated. For example, function self-time can be calculated. Self-time is the time spent in the body of a function excluding any time spent in the functions it calls. Self-time can be calculated by subtracting the cumulative time spent in the function's children from

the cumulative time spent in the function itself. In addition, the average time per function call can be calculated by dividing call time by the call count.

5.1.3 Call Arcs

WasmProf takes the approach of only recording information about call arcs. A call arc is made up of a call source and a call destination. For each unique source, destination pair, WasmProf keeps a counter of the number of times that call arc was traversed and how much time was spent in the call arc. Time spent in a call arc is the time between when the function call is made and when the function call returns. A call arc traversal is any runtime function call that originates at the source and calls the destination function.

This approach of tracking call arcs is a similar design to that of the gprof profiler [7]. The disadvantage to only tracking call arcs, and not a complete call graph, is that the exact call graph cannot be recreated in all cases. Rather, the call graph is built using the call arc information. This data tells exactly which functions called which other functions, but the exact path of a given function call cannot always be determined. For example, take the case where `func3` calls `func5` only when called from `func1`, and `func3` calls `func4` only when called from `func2`. The exact structure of this call graph cannot be determined using call arcs. The best that can be recreated is that both `func1` and `func2` call `func3`, and `func3` calls both `func4` and `func5`. In practice, this is not too limiting and the call graph produced should still be useful to the developer.

On the other hand, there are a number of advantages to using call arcs. First, it is faster since the profiler doesn't have to "walk" the stack or call context to determine the call graph. Second, less space is required to store call arc information than to store a complete call graph. This is the most important advantage for WasmProf because,

as discussed in Section 5.2, WasmProf needs to be able to statically determine how much space it needs for profiler data at instrumentation time.

5.1.4 Host API

WasmProf defines a number of API functions that must be implemented by the host environment and made available for calling within WebAssembly. The WasmProf tool generates these function implementations automatically into a supporting JavaScript file when it is run. However, WasmProf instrumented programs should run correctly in any host environment that implements this API.

All functions must be made available by the host environment and will be imported by a WasmProf instrumented binary under the "prof" module. See Appendix A for an example of how the instrumented WebAssembly imports these functions. The required functions are:

- float getTime(void): This returns the current time as a floating point number. The only requirement on the getTime() function is that it uses a monotonically increasing clock.
- void clearResults(void): This clears all arc data stored in host environment data structures.
- void addArcData(int srcID, int destID, float callCount, float targetTime): This provides arc data that should be added to arc data sent with previous and subsequent calls to addArcData (i.e., all calls to addArcData with the same source, destination pair should be added together). It is on the host environment to create the data structures to store this data. This is used to support the special case when WasmProf cannot store all data internally, as described in Section 5.3.3.
- void setArcData(int srcID, int destID, float callCount, float targetTime): This provides arc data that is complete and should overwrite existing arc data for

the given source, destination pair. It is on the host environment to create the data structures to store this data until clearResults is called.

- void printResults(): This allows the WebAssembly code to dictate when results are printed, if desired. This will be called after all profiling data is sent to the host using the addArcData and setArcData functions. The host should format and print the profiling data when this function is called. The main use case for this is when a WebAssembly program is run from the command line. In the browser, it makes sense to print the results from the interactive JavaScript console.

5.2 Limitations of WebAssembly

WasmProf was implemented to work with programs compiled for the WebAssembly MVP (version 1.0) specification [14]. This comes with a number of limitations that influenced design decisions made during the creation of WasmProf.

5.2.1 Memory Limitation

WebAssembly MVP defines a WASM program to have a single linear memory which is either declared in a WASM module or imported into a WASM module [14].

In the case of imported memory, the memory could be modified by code outside of what is being considered by WasmProf. So WasmProf cannot determine what memory would be safe to use for its own data structures.

Since WasmProf cannot determine how the original program will use the dynamic memory, WasmProf must avoid using the linear memory entirely. This means that all data structures must be statically allocated in globals at the time the original program is being instrumented. This works because, in WebAssembly, globals exist separately from the linear memory. The memory limitation means that WasmProf cannot dynamically allocate itself extra space which is a requirement for building a

data structure that fully tracks a complete execution tree. To get around this, WasmProf tracks call arcs as specified in Section 5.1.3. By traversing the WebAssembly code, WasmProf can determine the set of all possible call arcs and preallocate space to store information about each arc. This does waste space since some arcs will never be called, but it is the only solution to tracking call information without relying heavily on the host environment (i.e., calling out to JavaScript for every update as Wasabi does [10]).

5.2.2 Access to System Resources

WebAssembly does not have inherent access to system resources that are required to implement a full profiler. Most notable, WebAssembly has no timer and has no access to I/O without relying on function imports from the host environment. For this reason, WasmProf cannot possibly be implemented completely in WebAssembly and must rely on the host environment to provide the timer and access to I/O so results can be printed. This led to the design of the host API previously described (in Section 5.1.4).

5.3 Instrumentation Implementation

WasmProf was implemented in C++ using the Binaryen compiler toolchain [2]. Binaryen provides the WebAssembly binary parsing and outputs an AST style data structure. That structure is then modified in-place to add instrumentation code while still preserving the original execution of the program. (i.e., the instrumented program will always produce the same results as the original program).

Instrumentation is added by traversing the AST depth-first and looking for every call instruction. For blocks of expressions where every expression is at the same AST depth, the expressions are traversed in the order they would execute. This ensures that function calls are encountered in exactly the order they would execute at runtime,

which is important for maintaining correct function execution when the function calls are moved around (see call hoisting below).

In WebAssembly, calls consist of both the `call` and the `call_indirect` instructions. A `call` is simple because both its source and destination functions can be resolved statically when doing the instrumentation. Indirect calls, which index into a table of function pointers, require more work to resolve correctly. The possible implementations for indirect calls and the solution WasmProf uses are discussed in Section 5.5.2 below.

Before and after every call instruction, a call to the getTime() function is made so that time in the function can be calculated. If the WasmProf option to enable forced printing is specified, then on exit from a top-level function (i.e., a WASM function that was called from JavaScript without originating from WASM) the printResults() function will be called.

Assuming the following call sequence:

$jsF1(js) \rightarrow main(wasm) \rightarrow jsF2(js) \rightarrow wasmF1(wasm) \rightarrow wasmF2(wasm)$

Only the *main* function would be considered a top level WebAssembly function. So printResults() would be called right before *main* returns to $jsF1$.

5.3.1 Call Hoisting

In order to add instrumentation code surrounding every function call, WasmProf needs to split apart nested calls while maintaining correct code execution order and side-effecting. This is done by traversing the AST depth-first and hoisting every call into the nearest block above it in the AST. If the result of a hoisted function is not void, then its returned value is stored in a local variable and used wherever the function was previously located.

Hoisting helps avoid some unnecessary nesting of timing overhead. If the timing instrumentation were added in-place, then the timing for the outer function call would

include the extra calls to getTime for the inner function call. Hoisting the nested function calls so they are called one after the other in the same block avoids the timing instrumentation being nested. Note that it is still not possible to avoid the timing overhead when a function calls another function in its body, but WasmProf tries to reduce the overhead from timing instrumentation wherever possible which hoisting helps with. Figures 5.1 and 5.2 show what a nested function call looks like before and after hoisting.

Example nested call (before):

```
1 if(cond){
2     function1(12, function2(23,1));
3 }
```

Figure 5.1: Example nested call before hoisting and instrumentation

Example call hoisting (after):

```
1 if(cond){
2     var f2_start_time = getTime();
3     var f2_temp = function2(23,1);
4     global f2_total_time += (f2_start_time - getTime());
5
6     var f1_start_time = getTime();
7     function1(12, f2_temp); //already in block, doesn't need hoisting
8     global f1_total_time += (f1_start_time - getTime());
9 }
```

Figure 5.2: Example nested call after hoisting and instrumentation

5.3.2 WebAssembly Support Functions

In addition to instrumenting all calls, WasmProf adds a number of helper functions to the WebAssembly binary. These helper functions are called when the WebAssembly is

done executing and facilitate extracting all of the call arc information to the host. On exit from a top-level WebAssembly function, a helper function is called that invokes the setArcData API function for every arc stored in WebAssembly. This can take time but does not interfere with the profiler results at all because it is only called at the very end after all times have already been calculated.

5.3.3 Decorators

All functions for which call source and destination cannot be determined at instrumentation time are handled by using function decorators. The function decorator is called in place of the original function and examines a lastCaller global variable that was previously set to signal what the source of an unknown call arc is. How the lastCaller global is set is discussed Section 5.5 as it varies depending on how the decorator is being called. With the lastCaller as the source and the decorated function as the destination, WasmProf can determine which call arc to use. However, this determination must be done at runtime whereas normal calls can resolve their call arcs at instrumentation time. There are two different strategies used for making this runtime determination of the call arc.

The first strategy is to use a switch in the decorator and to use the lastCaller as the switching variable. This works well when there are relatively few cases for lastCaller, which is the case when the decorated function can only possibly be called from a handful of places. The advantage of this strategy is everything is resolved in WebAssembly and it should be fast as long as there are not many possibilities. The disadvantage is the length of the switch if there are a lot of possibilities for lastCaller. The long switch adds a lot of computational overhead and also increases the size of the binary dramatically if there are many possibilities. In addition, WasmProf would need to statically allocate globals for every possible source-destination pair. This is a

worst-case n^2 storage and searching complexity for n functions if it is possible to call every function from every other function.

The second strategy is to push the work of resolving the call arc off to the host. This is done using the `addArcData` host API function. The source argument can be set to lastCaller, the destination argument is set to the decorated function, call count is set to 1, and time is calculated by adding calls to `getTime()` before and after the call to the decorated function. This strategy works better when there are a lot of possibilities for lastCaller. The advantage here is that the host environment can use dynamic data structures and won't have to use a long switch.

In either case, the source and destination of a call arc can be correctly resolved at runtime.

5.4 Host Implementation

WasmProf requires that some amount of glue code is implemented by the host. This fills the gaps in WebAssembly's capabilities that are required to make the profiler work. The host glue code for this thesis is implemented in JavaScript since that is the most common host environment for WebAssembly modules. WasmProf automatically generates and outputs this accompanying JavaScript code when instrumenting a WebAssembly binary. The accompanying JavaScript serves a few different purposes; it overrides WebAssembly instantiation to provide host API implementations, stores call arc data, and facilitates printing profiler results in a readable format.

5.4.1 API JavaScript Implementation

The first thing the host JavaScript code does is implement the host API functions WasmProf needs. Below are the JavaScript implementations created for the API described in Section 5.1.4.

- float getTime(void): Implemented using JavaScript performance.now() function

- void clearResults(void): Simply clears JavaScript call arc data structure
- void addArcData(int srcID, int destID, float callCount, float targetTime): Checks if an entry for the srcID, destID pair exists in the data structure and creates entry if required. Increments the call count and time stored by the amount specified in the function arguments.
- void setArcData(int srcID, int destID, float callCount, float targetTime): Checks if an entry for the srcID, destID pair exists in the data structure and creates entry if required. Sets the call count and time stored to be the amount specified in the function arguments.
- void printResults(): Does processing and prints profiler results as specified in Section 5.4.4.

5.4.2 Runtime Patching

Once the host API functions are implemented they need to be made available to the WebAssembly code. This is done by adding the function references to the `WebAssembly.Module` object that is created during WebAssembly instantiation in JavaScript. The patching code does this in a way that does not interfere with how the original JavaScript code may be instantiating and interacting with a WebAssembly module. WasmProf handles this by runtime patching (monkey patching) the WebAssembly instantiation functions. The patched instantiation function sets up the WebAssembly module so that the host API implementations are mapped to the WebAssembly imports. The API implementations are added to the set of function implementations that are passed into the patched instantiation function. This complete set of function implementations should cover all the functions the instrumented WebAssembly program is expecting to import. Finally, the patched instantiation function calls the original instantiation function, passing it the complete set of function implementations.

5.4.3 Data Structures

In addition to implementing functions, the WasmProf JavaScript code also creates a data structure for storing arc information and a data structure for storing function names. The arc information structure is basically a hash map that maps a source, destination pair to an object containing an integer call counter and a float time variable for the call arc. The entries in this data structure are created as needed whenever one of the `addArcData` or `setArcData` API functions is called from within WebAssembly. The whole structure is deleted whenever the clearResults API function is called.

WasmProf internally tracks every function by assigning it a numeric ID. So the source and destination functions in a call arc are actually represented using the function ID. WasmProf doesn't store the function names in WebAssembly; rather, it builds a look-up table that maps a function ID to a function name and it outputs that to the accompanying JavaScript glue code. This way, when the WebAssembly function completes and wants to print collected profiler data, the JavaScript host can use this lookup table to print meaningful function names.

5.4.4 Printing Results

WasmProf results can be printed in one of two ways. The first way is for the WebAssembly code to initiate a print via the printResults API call. WasmProf accepts a command line flag that will cause the generated WebAssembly binary to call this print function whenever it exits from a top-level WebAssembly function. The second way to trigger a print is to call the WasmProf print function from within JavaScript. In either case, the results that are printed are exactly the same.

The print function itself first does some processing to transform the list of call arcs into times and counts for each function (rather than call arc). Each function has three main data points:

- called: How many times the function was called.
- cumulative time: How much total time was spent in the function. Including in functions it calls.
- self time: How much time was spent in just the function itself, not including time spent in functions it calls.

Using this data, along with the call arc information, WasmProf generates two main profiler outputs. The flat profile and the call graph profile, example of which are shown below in Figures 5.3 and 5.4. The format of these outputs is designed to match the familiar output of gprof [7].

%	cumulative	self	called	self	total	name
				ms/call	ms/call	
90.48	0.042	0.038	1	0.038	0.042	_main
7.14	0.003	0.003	1	0.003	0.003	_double
2.38	0.001	0.001	1	0.001	0.001	_add

Figure 5.3: Flat profile of a main function that makes one call each to add and double functions.

```
index   % time   self    children        called   name
                                                    <spontaneous>
[0]     100.00   0.038   0.004           1        _main
                 0.001   0.000           1/1      _add [2]
                 0.003   0.000           1/1      _double [1]
-----------------------------------------------------------
                 0.003   0.000           1/1      _main [0]
[1]     7.14     0.003   0.000           1        _double
-----------------------------------------------------------
                 0.001   0.000           1/1      _main [0]
[2]     2.38     0.001   0.000           1        _add
-----------------------------------------------------------
```

Figure 5.4: Call graph profile that shows main calling add and double functions.

5.5 Special Call Situations

There are two instances in which the call arc cannot be statically determined at instrumentation time. These are the cases when a decorator must be used to determine the call arc at runtime: calls to exported functions and indirect calls.

5.5.1 Calls to Exported Functions

WebAssembly code may call a function that is implemented in the host environment and the host environment could then potentially call back into WebAssembly. The example in Figures 5.5 and 5.6 shows how the call arc from `wasmMain`→`jsFunction` (call from WebAssembly to imported function) can easily be determined, but the call arc from `jsFunction`→`wasmFunction` (call from host environment to WebAssembly exported function) cannot be predetermined.

To handle host to WebAssembly calls, the lastCaller global variable is used along with a decorator as described in Section 5.3.3. The export in WebAssembly will be

replaced with a decorator, so the decorator gets called whenever the host environment calls that exported function. The lastCaller global is set right before making the call to jsFunction. The value of lastCaller will be the ID of jsFunction. Either jsFunction will return without calling back into WebAssembly in which case lastCaller is ignored, or jsFunction will call into the decorator set up for the exported function and the call arc will be resolved.

WasmProf uses the first decorator strategy from Section 5.3.3 for these exported function decorators since generally there are only a handful of exported functions and it can be determined exactly which functions they could possibly come from at instrumentation time. So the set of possible values for lastCaller is relatively small and WasmProf can take advantage of keeping the code in WebAssembly.

There is one special situation when JavaScript initially makes a call into WebAssembly. The lastCaller global will not have been set. To handle this, lastCaller is assigned the special initial value of 0 which means that the source was "spontaneous" (i.e. it was the start of execution).

```
1 func jsFunction(){
2     ...
3     wasmModule.wasmFunc(1, 2, 3); //redirect to wasmFuncDecorator
4     ...
5 }
```

Figure 5.5: Example JavaScript code that calls a WebAssembly function

```
1  import "jsFunction" jsFunction
2  export "wasmFunc" wasmFuncDecorator //was originally wasmFunc
3  wasmMain{
4      ...
5      set lastCaller = jsFunction_ID
6      //start timing
7      call jsFunction
8      //end timing
9  }
10 wasmFuncDecorator(parm1, parm2, parm3){
11     switch(lastCaller):
12         ...
13         case jsFunction_ID: //call arc must be jsFunction->wasmFunc
14         ...
15     end_switch
16     //start timing
17     call wasmFunc(parm1, parm2, parm3)
18     //end timing
19 }
```

Figure 5.6: Example WebAssembly code that imports and calls a JavaScript function

5.5.2 Indirect Calls

Call arcs cannot be determined at instrumentation time when an indirect call is made. An indirect call indexes into a function table to determine which function to call, and WasmProf cannot determine at instrumentation time what the value of that index will be.

WasmProf again uses a decorator function to resolve this problem. Every entry in the function table is replaced with a corresponding decorator function, so when `call_indirect` is called it will actually call the decorator function. To convey the

source of the call arc the lastCaller global is used again; right before the indirect call is made the lastCaller variable is set to the current, calling function (which is different than in the case of an imported function call). Now in the decorator, the call arc can be determined using lastCaller as the source and the decorated function as the destination.

WasmProf uses the second decorator strategy from Section 5.3.3 for indirect calls and resolves the arc by relying on the host environment. This strategy works better for indirect calls because each `call_indirect` could potentially call any of the functions in the function table that are of the correct type. This means that the set of possible values for lastCaller is every function that contains a `call_indirect` instruction.

Chapter 6

RESULTS

6.1 Experimental Setup

The following sections describe the experimental setup used to collect WasmProf performance metrics. The setup consists of other profilers to which WasmProf is compared, WebAssembly benchmarking programs on which the profilers are tested, and the characteristics of the physical machine the tests were conducted on.

6.1.1 Profiler Comparisons

WasmProf was compared to Wasabi by implementing an equivalent profiler using the Wasabi framework [10]. For the purpose of creating a profiler, the whole set of dynamic analysis hooks provided by Wasabi were not needed, so only call instruction hooks, begin hooks, and return instruction hooks were kept. These hooks cause the Wasabi instrumented binary to call out to a JavaScript function before and after each call instruction. This allows for profiling information to be collected and stored in those JavaScript functions, which are implemented as part of this thesis. The initial implementation using Wasabi relied a lot on JavaScript's dynamic language features, such as adding fields to objects. This turned out to be very slow, approximately an order of magnitude slower than the WasmProf overhead on some tests. The JavaScript for the Wasabi profiler was rewritten in a manner that was friendlier to the JavaScript compiler so it would perform better (this code is listed in Appendix C) and the implemented Wasabi profiler was used for comparison to WasmProf below.

Additionally, WasmProf was compared to the sampling profilers built into Chrome and Firefox. The performance of the sampling profilers was expected to exceed that of both WasmProf and Wasabi, but the comparison is provided as a baseline for how much overhead sampling profilers add to the WebAssembly execution.

6.1.2 Benchmark Programs

Performance results are obtained by instrumenting and running the following WebAssembly programs.

The first set of benchmark programs come from an open source set of simple WebAssembly programs designed to test the speed of WebAssembly against JavaScript [19]. All of these programs were originally written in C and are compiled to WebAssembly using Emscripten [21] (Emscripten was originally a LLVM to JavaScript compiler and is now the defacto standard for compiling C/C++ to WebAssembly/-JavaScript)[1]. Two main compiler options were specified when compiling the benchmarks: `-Os` to run compiler optimizations and `-g` to include debugging information such as function names which are used by WasmProf (the `-g` flag does disable some of the optimizations that would normally be run with `-Os`). Some of the similar benchmarks in this set are grouped together and their results averaged for brevity.

The first group of benchmarks consists of simple math and array operation kernels. This group includes MultiplyInt, MultiplyDouble, MultiplyIntVec, MultiplyDoubleVec, QuicksortInt, QuicksortDouble, SumInt, and SumDouble.

The second group of benchmarks consists of image manipulation kernels. This group includes ImageConvolute, ImageGrayscale, ImageThreshold.

The third group is actually a single benchmark called Video Marker Detection. The marker detection program checks each frame in a video for a specific marker image and figures out the position of the marker so it can be outlined. This program is a little more complicated than the simple math and image processing kernels, and therefore is a better real-world test of WasmProf's overhead.

The final benchmark program comes from the open source GameBoy emulator WasmBoy [17]. It is written in Typescript and compiled to WebAssembly. This is the most complicated program WasmProf was run with and serves as the best

[1] https://emscripten.org/docs/compiling/WebAssembly.html

real-world example of where a profiler like WasmProf might be used. Performance numbers for WasmBoy were collected using a benchmarking setup included in the repository. Results were collected with a configuration that runs 1000 frames of an emulation and measures how long it takes to render each frame.

6.1.3 Test Machine

All WebAssembly programs were tested on a Windows 10 computer running an i7-6700k processor and 16GB of RAM. Programs were tested using the stable build of Chrome version 74.0.3729 and the stable build of Firefox version 67.0.0. By default, both Firefox and Chrome reduce their timer precision to prevent fingerprinting. In Firefox this can be disabled to give Firefox a timer precision of 1 microsecond. Chrome does not have a configuration for timer precision that we could find; it has a timer accuracy of 100 microseconds with some pseudo-random jitter added [2]. This affects the timing of the profile a little bit, but results relating to the overhead of the profilers in Chrome are still valid because the WebAssembly programs tested take orders of magnitude more time to execute than the timer precision floor.

6.2 Validation Results

We validated that both WasmProf and the equivalent Wasabi profiler met the requirements in Section 4 for an instrumenting profiler. Neither profiler interfered with the execution of the original program and both correctly tracked every function call as expected.

6.2.1 Concrete Requirements

To verify the first concrete requirement, that the instrumented program produces the same output as the uninstrumented program, the set of open source bench-

[2] https://chromium-review.googlesource.com/c/chromium/src/+/849993

mark programs mentioned in Section 6.1.2 were used. Importantly, all of the benchmarks check the result from their executed WebAssembly program against a reference JavaScript implementation. As expected, both the uninstrumented and instrumented WebAssembly programs match the reference implementation for all of the benchmarks. WasmProf (and the Wasabi profiler) were also run on a more complicated application; the GameBoy emulator WasmBoy[17]. The emulator benchmark still behaved correctly through 2500 frames of emulation using an instrumented WebAssembly file.

To verify the requirement that all functions are tracked a number of simple, "toy" WebAssembly programs were created with a known exact call path. These programs were instrumented and manually verified to ensure all functions were being tracked. Additionally, when comparing WasmProf against a profiler implementation using Wasabi, the output from the above mentioned WebAssembly Benchmarks were examined and certain functions were spot checked to verify that the number of function calls matched.

6.2.2 Functional Requirements

The first functional requirement is to keep overhead from instrumentation low. This requirement was evaluated by comparing the performance of WasmProf to the performance of the original code, browsers' sampling profilers, and a Wasabi profiler. The results from the performance comparison are detailed in Section 6.3.

The second functional requirement is to ensure function-level timing was accurate relative to the contents of each function. To validate this, a simple test program was created with functions whose relative run time is predictable. The code for this and the corresponding WasmProf output can be found in Appendix B. Overall, the test showed that WasmProf behaved as expected. There is some slight variation in function timings, but the variation was within a 0.7% difference of the expected

relative function times. In addition, other programs such as the marker detection benchmark were spot checked to ensure a few of the function times made sense in the context of when they were being called. The top-level function was checked along with a few of its children and no irregularities in the timing were found. Further rigorous analysis of function timing accuracy is left for future work.

6.3 Performance Results

6.3.1 Binary Size Difference

Program	Original Binary	WasmProf Binary	Wasabi Binary
Simple Math (avg)	110.8	212.0	218.7
Image Processing (avg)	111.3	212.7	219.2
Marker Detection	131.4	245.8	248.5
wasmBoy	35.4	79.8	70.2

Table 6.1: Binary size of original program and instrumented programs (in kilobytes)

From the data, it is clear that WasmProf produces instrumented WebAssembly binaries that are approximately twice the size of the original program's binary. Additionally, the accompanying JavaScript adds approximately another 6 kilobytes plus the size of the function ID to name map. This is about the same binary size overhead as Wasabi produces and is within acceptable overhead for WasmProf. This overhead is considered acceptable because the binary size has little effect on runtime performance; it may just slow down initial start-up slightly as the larger file is downloaded. That is not a major performance concern since an instrumented program would never be used in a production environment where start-up latency is crucial.

6.3.2 Compilation Difference

The compilation time of the WebAssembly binaries was measured using the JavaScript call

```
var module = new WebAssembly.Module(buffer);
```

which synchronously compiles the binary WebAssembly stored in the buffer argument. The generated module object contains the compiled WebAssembly code in a stateless form, meaning it is compiled but has not mapped imports and exports. Compilation time was measured in Firefox 67.0 200 times for each WebAssembly file and averaged to get the following results. Chrome does not allow for synchronous WebAssembly compilation for files over 4kB so it was not used for this test.

Program	Original Binary	WasmProf Binary	Wasabi Binary
Simple Math (avg)	4.82	45.32	40.61
Image Processing (avg)	4.81	46.37	40.62
Marker Detection	6.09	50.26	43.52
wasmBoy	6.24	16.35	11.43

Table 6.2: Compilation time of WebAssembly program (in milliseconds)

Even with the compilation overhead, the worst WasmProf binary only took 50ms to compile. Since WasmProf is meant to be used for testing and analysis and not in a production application, this small amount of extra start-up overhead does not take away from WasmProf's functionality.

6.3.3 Runtime Difference

Program	Original Binary		WasmProf Binary		Wasabi Binary	
	Firefox	Chrome	Firefox	Chrome	Firefox	Chrome
Simple Math (avg)	100%	81.5%	174.1%	111.2%	411.5%	953.2%
Image Processing (avg)	100%	91.7%	108.2%	92.4%	240.6%	486.4%
Marker Detection	100%	75.6%	767.3%	324.3%	1201.6%	2171.5%
WasmBoy	100%	107.8%	12127.4%	5114.0%	14711.8%	19821.5%

Table 6.3: Runtime of benchmark programs. Normalized to the runtime of the original WebAssembly binary in Firefox.

The performance overhead of WasmProf varied widely from application to application, but some characteristics of programs that may cause WasmProf to have more overhead can be deduced from the data.

The first thing to notice is that the simple math benchmarks and the image processing benchmarks have low overhead when instrumented by WasmProf. This is due to the fact that these functions do not have very deep call graphs and do most of their work in a few top-level functions. This reduces any extra overhead that may arise from nested function timing. On the other hand, more complex benchmarks like Marker Detection and WasmBoy, generally have much higher overhead likely due to the fact that they have deeper call graphs and are executing a lot more function calls. This is further supported by the fact that benchmarks with a lot of recursion (quicksortDouble, quicksortInt) had higher overhead than simple math benchmarks. Specifically, the WasmProf instrumented quicksortDouble had a runtime of 400% on Firefox and 200% on Chrome compared to the original binary running in Firefox. Additionally, a recursive Fibonacci benchmark had to be excluded from these results because it took too long to run when instrumented. This particular benchmark was implemented recursively and therefore consisted entirely of a lot of very small function

calls to calculate Fibonacci numbers. Programs such as this, with a lot of small functions, suffer from the overhead of getting the start and end times for every single short function call. In an isolated test, the performance of the `getTime` function was measured and found to take about 350 nanoseconds and 100 nanoseconds per call for Chrome and Firefox respectively. Adding such overhead to a function call that takes a fraction of that time to execute will naturally lead to high overhead and possibly inconsistent timings. So in general, WasmProf (and the Wasabi profiler) work much better on programs that do not create deep call stacks frequently and that have fewer and longer running functions.

A second important takeaway from the performance results is that WasmProf introduces less overhead than does the Wasabi profiler. This validates that the design decisions made to improve the performance of WasmProf actually made a difference. Namely, the choice to track timing data structures as much as possible in WebAssembly seems to make a difference over the strategy of always calling out to the host environment to manage data, as Wasabi does. Figure 6.1 shows a graphical comparison of the WasmProf and Wasabi overhead on the WasmBoy benchmark. The small bars show one standard deviation above and below the average time to render a frame. The graph shows that the performance improvement of WasmProf is statistically significant.

Another interesting thing to note is the wide variance in performance between Firefox and Chrome. On the WasmBoy and marker detection benchmarks Chrome runs the WasmProf instrumented program about two times faster than Firefox. However, in the image processing benchmarks, they are very similar. Furthermore, Firefox seems to run the Wasabi profiled programs faster than Chrome in pretty much every case. These differences are likely caused by the disparity in how Chrome and Firefox perform optimizations in their respective JIT compilers. There was no clear pattern

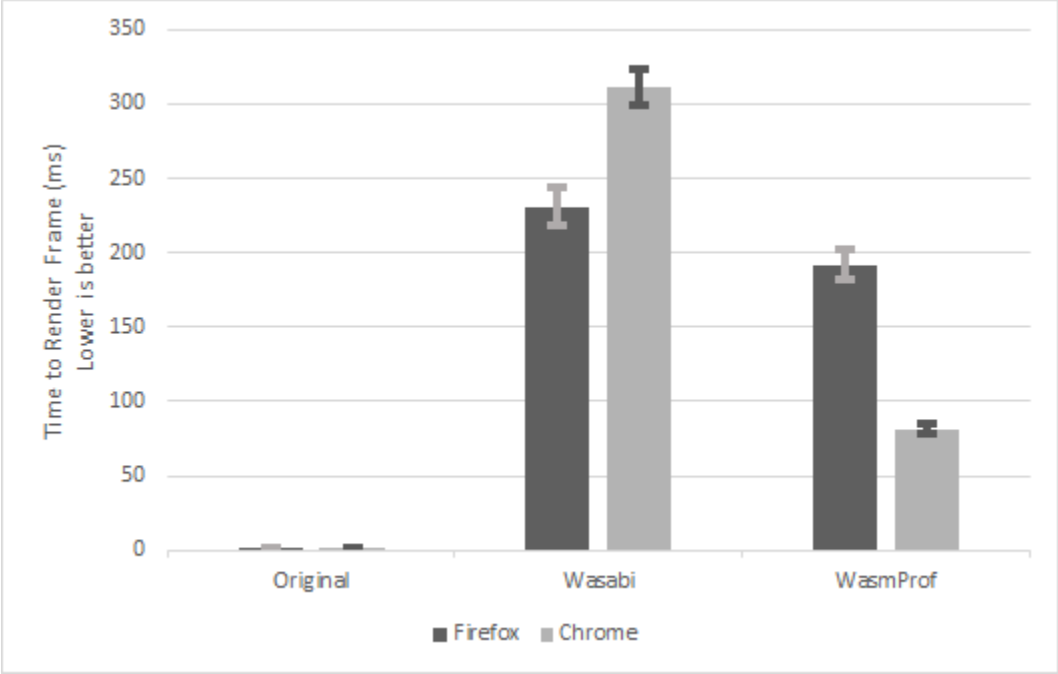

Figure 6.1: WasmBoy performance comparison of WasmProf, Wasabi, and unmodified program

on what made some runs faster in a specific browser, but in general, Chrome seemed to work better with WasmProf instrumented programs.

Figure 6.2 shows the performance of the WasmBoy benchmark when running the original program compared to running the original program with the browser's sampling profiler running. The sampling profiler seems to have a slight performance cost, but nothing that is statistically significant. This does make sense since, as noted in Section 2.3.2, the browser's runtime is actually always sampling the performance of executing code so it can decide what to optimize further. WasmProf and other instrumenting profilers cannot get anywhere near the low overhead of the sampling profiler. As previously noted, a good analysis workflow will likely include the usage of both the sampling profiler as well as a tool like WasmProf, since they are good at different things.

Note, these performance results show how much overhead is introduced when using WasmProf, but these results do not directly affect the accuracy of WasmProf's

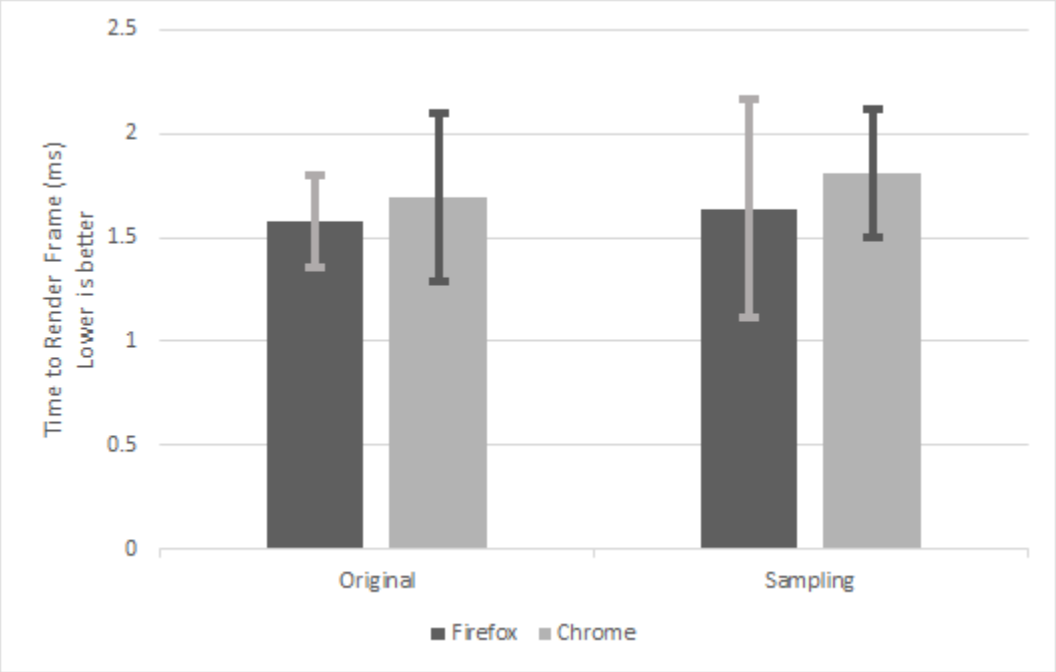

Figure 6.2: WasmBoy performance comparison of sampling profiler and unmodified program

relative function timings. Given the high overhead, WasmProf cannot be expected to produce function time data that is exactly accurate to the original program; however, as discussed before what is important is the relative timing data of functions compared to each other. These relative function times can still give a picture of the program's execution even if they are not exactly accurate to the original program. That being said, the performance overhead can skew the function timing results some since the timing of functions near the top of the call graph may be affected by the timing overhead of functions lower in the call graph. This effect is difficult to quantify and a more rigorous quantitative analysis is left to future work.

6.4 Usability and Output

The ease of use of WasmProf is a strong point of the profiling tool. Sections 6.4.1 and 6.4.2 outline the steps to instrument a WebAssembly binary and collect profiling

information at runtime in order to illustrate how easy it is for a user to collect data with the WasmProf tool.

6.4.1 Instrumentation Usage

Instrumenting a WebAssembly program using WasmProf is an easy process that only requires a few commands. The instrumentation procedure is as follows

- Run WasmProf on the original WebAssembly file. WasmProf will create two output files: an instrumented WebAssembly binary and a supporting JavaScript file.
- Replace original WebAssembly file with the instrumented binary.
- Either include the produced JavaScript file in a `<script>` tag before JavaScript code that instantiates WebAssembly or just prepend the contents of the produced JavaScript file to the beginning of the existing JavaScript.

One thing to note is WasmProf works better with WebAssembly files that have a Names custom section as mentioned in Section 2.3.1. This allows the profiler to print out meaningful function names rather than numeric function IDs.

6.4.2 Runtime Usage

Once a WebAssembly binary is instrumented, runtime usage of WasmProf is very simple as well. The user just runs the program as usual in the browser after including the instrumented WebAssembly file and the supporting JavaScript.

Once the program is running, profiling data can be accessed using the browser's JavaScript console. First, the following command needs to be run to save a snapshot of the profiler data collected until that point.

```
var results = window.WasmProf.saveResults();
```

The returned JavaScript object contains all the information to print profiler data. The call graph output can be printed with the command `results.callGraph();`

and will produce an output like the one in Figure 5.4. Similarly, the flat profile can be printed with the command `results.flatProfile();` and will produce an output like the one in Figure 5.3. In either case, obtaining profiler data is simple for the user and could even be scripted to take snapshots of the profiler data at regular intervals. This scheme works well for both single run WebAssembly programs as well as WebAssembly programs that can be run continuously (the WasmBoy emulator can be run continuously but was tested as a single run of 1000 frames).

6.4.3 Output Inaccuracies

In a few of the results, there were inaccuracies in the value reported for self-time on some functions. An example of such an inaccuracy is shown in Figure 6.3, which shows a partial call graph of the Marker Detection benchmark program. The function `findMarkers` incorrectly reports a negative self-time.

```
ind  % time   self   children   called      name
                                            <spontaneous>
[0]  99.99    3.000   16775.0    539         _detect
              3.000   0.000      539/539     _freeMarkers [45]
              5.000   16763.000  539/539     _ARDetector::detect) [1]
-------------------------------------------------------------
              5.000   16763.000  539/539     _detect [0]
[1]  99.93    5.000   16763.000  539         _ARDetector::detect
              232.0   0.000      539/539     _CV::grayscale [20]
              -309.0  968.000    539/539     _ARDetector::findMarkers [13]
 -------------------------------------------------------------
              -309.0  968.000    539/539     _ARDetector::detect [1]
[13]  3.93    -309.0  968.000    539         _ARDetector::findMarkers
              794.917 1831.531   579/652595  __push_back_slow_path [6]
              128.321 34.219     1537/1537   _CV::warp [27]
```

Figure 6.3: Example of inaccurate timing output.

The reason this happens is that WasmProf is making assumptions about the behavior of the program that may not always be true; this can sometimes lead to strange and incorrect timing results. When estimating how much time is spent in a child function, WasmProf assumes that the execution time of that child function does not depend on where it was called from. So if child function `c_func` is called from parent function `p_func` 5 times, and `c_func` is called a total of 10 times (from anywhere) with a cumulative time of 10 seconds, then it is reasonable to estimate that 5 of those seconds can be attributed to the children of `p_func` . However, this is not necessarily true and can skew the calculation of `p_func`'s self-time. So the timing inaccuracy happens when a function is called from multiple places and the profiler must estimate how much of the function's overall time to attribute to each of those calls. This case will never occur in programs where a function is never called from multiple other functions (i.e., when a function only ever has one parent in the call graph).

These inaccuracies do take away from WasmProf's ability to meet the relative timing accuracy functional requirement but not to a great extent. The inaccuracies don't occur on most functions, and in all cases, the cumulative time for each function is still correct. With this information, a user can deduce why the timing may be inaccurate (called from multiple places) and look at other portions of the data like cumulative time to determine what is going on in the program. The inaccuracy is a downside to using WasmProf, but no good solution exists under the current design. Gprof suffers from a similar problem [7][3]. Collecting more complete profiler data rather than call arcs is the only way to completely avoid this problem (see Chapter 7).

[3]https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_chapter/gprof_6.html

Chapter 7

FUTURE WORK

7.1 Profiler Improvements

There are a number of profiler features that are out of the scope of this thesis but that may be interesting to explore in the future. One such feature is to intelligently detect small functions and remove some or all of the instrumentation from those functions. This has a potential benefit of eliminating some of the profiler's overhead since the instrumentation overhead to function body ratio is very high for small functions. As an example, if a small add function were called millions of times deep in the call graph, that would add a lot of overhead to all the function timings higher up in the call graph and may skew results. Tracking the add function is probably not important so removing its instrumentation would reduce the overhead. Detecting these types of small functions could either be done manually by the user, automatically at instrumentation time, or dynamically at runtime.

Another potential feature would be call graph splitting. This would allow the profiler to differentiate call arcs not just based on source and destination, but also based on the parameters passed to the call instruction at runtime. This adds more information that the user can use to analyze the execution of their program. However, this adds a lot of runtime complexity to the profiler and will likely result in even higher overhead. Future work could explore the advantages and disadvantages of features such as this.

A final potential area of research would be to explore further the differences in execution time between runtime engines seen in Section 6.3. It is possible that with a better understanding of how each engine optimizes code, the profiler could be tuned

to the specific nuances of each runtime and further reduce the overhead of the profiled code.

7.2 Profiler with Future WebAssembly Features

WebAssembly is still evolving and many of the details of the language that were limitations for WasmProf may change in the future. As WebAssembly develops further and adds features, more work can be done to add new profiling capabilities. The most significant change that could drive future profiler work would be allowing for multiple linear memories in a single WebAssembly module [1]. With multiple linear memories, a profiler could declare a memory section for its own use and make use of more sophisticated dynamic data structures for tracking profiling data, all without affecting the memory of the original program. This would enable additional features such as fully traceable call stacks, rather than just tracking individual call arcs. It would also allow for an implementation of tracking indirect calls that would be completely contained in WebAssembly. This would have the added benefit of eliminating unused globals for call arcs that are never called. It would likely reduce binary size overhead and it would be interesting for future work to examine how using WebAssembly memory vs WebAssembly globals affects profiler performance.

Another feature that is about to be added to WebAssembly is threading [2]. With the addition of threads, there is corresponding future work to be done in order for an instrumenting WebAssembly profiler to track function calls across every thread. This would add some complexity and race condition concerns to the profiler but should be able to be implemented in a similar manner as WasmProf.

Finally, as WebAssembly develops the runtime environments will get better and better. Improvements in runtime optimizations, faster WebAssembly to JavaScript

[1] https://github.com/WebAssembly/design/blob/master/FutureFeatures.md
[2] https://github.com/WebAssembly/threads/blob/master/proposals/threads/Overview.md

calls, and faster access to system resources such as timers could all aid in reducing the overhead of an instrumenting profiler such as WasmProf.

7.3 Other Host Environments

Another area of future development is extending WasmProf to work in environments other than JavaScript. The most obvious case would be a system runtime that does not require any supporting code and just runs pure WebAssembly. There is active development on a runtime call WasmTime that enables this [18]. WasmTime also supports the WebAssembly System Interface (WASI) that standardizes an API for system calls from WebAssembly [3]. Using a runtime that supports system calls, it may be possible to create a version of WasmProf that relies only on the system interface API and does not need to implement the host API as in this thesis.

[3]https://github.com/CraneStation/wasmtime/blob/master/docs/WASI-overview.md

Chapter 8

CONCLUSION

This thesis presented WasmProf, an instrumenting profiler for WebAssembly. WasmProf seeks to fill a gap in the spectrum of tools available for analyzing the performance of WebAssembly applications. Sampling profilers exist in all major browsers to support analyzing the performance of both JavaScript and WebAssembly code. These built-in profilers are readily available and have very little performance overhead, but they do not capture information about how many times a function is called and their statistical nature means it is possible to miss functions. On the other extreme, there has been some research into developing dynamic analysis tools for WebAssembly, most notably Wasabi [10]. Wasabi is able to instrument every WebAssembly instruction and does not miss any information. However, it relies entirely on calls out to the host environment to do anything useful with the information. This leads to high runtime overhead and requires the developer to include extra code to implement the Wasabi framework as a profiler. In the middle sits WasmProf, with the goal of instrumenting WebAssembly code to collect all information about function call counts and timing with as little overhead as possible while being easy to use.

The design of WasmProf was largely driven by this goal, as well as the necessity that it works within the constraints of WebAssembly's current specification. WebAssembly is only at version 1.0 and there are a lot of constraints (such as how memory is managed) that force WasmProf toward a design that tries to do as much work during the instrumentation phase in order to simplify the behavior of the instrumentation code during runtime. The resulting profiler collects information about call arcs and outputs data much like the famous profiler gprof [7].

In the end, results show that WasmProf is able to track all function calls as required. It has high overhead as expected for an instrumenting profiler, but in

general the runtime performance of WasmProf matched or beat an implementation of a similarly capable profiler written using the Wasabi framework.

WasmProf shows that it is possible to develop an instrumenting profiler for WebAssembly. We recognize that there is future work to be done as the WebAssembly specification and WebAssembly runtime environments develop further. WebAssembly tooling support is something that is still in its infancy and we look forward to seeing more tools like WasmProf develop in the future.

BIBLIOGRAPHY

[1] K. Basques and M. Kearney. Speed up javascript execution — tools for web developers — google developers, May 2019.

[2] Binaryen: Compiler infrastructure and toolchain library for WebAssembly, in C++, May 2019. https://github.com/WebAssembly/binaryen.

[3] W. Binder, J. Hulaas, and P. Moret. Advanced java bytecode instrumentation. In *Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 135–144. ACM, 2007.

[4] L. Clark. A crash course in just-in-time (jit) compilers mozilla hacks - the web developer blog, Feb 2017.

[5] L. Clark. What makes webassembly fast? mozilla hacks - the web developer blog, Feb 2017.

[6] W. Fu, R. Lin, and D. Inge. Taintassembly: Taint-based information flow control tracking for webassembly. *CoRR*, abs/1802.01050, 2018.

[7] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, pages 120–126, New York, NY, USA, 1982. ACM.

[8] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 185–200, New York, NY, USA, 2017. ACM.

[9] C. Hammacher. Liftoff: a new baseline compiler for webassembly in v8, Aug 2018.

[10] D. Lehmann and M. Pradel. Wasabi: A framework for dynamically analyzing webassembly. *CoRR*, abs/1808.10652, 2018.

[11] S. Liang and D. Viswanathan. Comprehensive profiling support in the java virtual machine. 05 1999.

[12] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[13] Profiling with the firefox profiler, Apr 2019. https://developer.mozilla.org/en-US/docs/Mozilla/Performance/Profiling_with_the_Built-in_Profiler.

[14] A. Rossberg. Webassembly core specification. W3C working draft, W3C, Sept. 2018. https://www.w3.org/TR/2018/WD-wasm-core-1-20180904/.

[15] P. Spiess and J. Swift. Webassembly: A new hope, 2017. https://pspdfkit.com/blog/2017/webassembly-a-new-hope/.

[16] M. Trivellato. Webassembly load times and performance, Sep 2018. https://blogs.unity3d.com/2018/09/17/webassembly-load-times-and-performance/.

[17] A. Turner. WasmBoy, May 2019. https://github.com/torch2424/wasmboy.

[18] Wasmtime: a WebAssembly Runtime., May 2019. https://github.com/CraneStation/wasmtime.

[19] WebAssembly-benchmark, June 2017. https://github.com/takahirox/WebAssembly-benchmark.

[20] J. Whaley. A portable sampling-based profiler for java virtual machines. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 78–87. Citeseer, 2000.

[21] A. Zakai. Emscripten: An llvm-to-javascript compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, pages 301–312, New York, NY, USA, 2011. ACM.

APPENDICES

Appendix A

EXAMPLE OF INSTRUMENTED WEBASSEMBLY IMPORTING REQUIRED FUNCTIONS

```
1    (type $t1 (func (result f64)))
2    (type $t2 (func))
3    (type $t3 (func (param i32 i32 i32 f64)))
4    (import "prof" "getTime" (func $prof.getTime (type $t1)))
5    (import "prof" "clearResults" (func $prof.clearResults (type $t2)))
6    (import "prof" "addArcData" (func $prof.addArcData (type $t3)))
7    (import "prof" "setArcData" (func $prof.setArcData (type $t3)))
8    (import "prof" "printResults" (func $prof.printResults (type $t2)))
```

Appendix B

PROGRAM TO VALIDATE RELATIVE FUNCTION TIMING OF WASMPROF

B.1 Validation Code

```
1  #define MAIN_ITERATIONS 10000000
2  #define HALF_ITERATIONS (MAIN_ITERATIONS/2)
3  #define QUARTER_ITERATIONS (MAIN_ITERATIONS/4)
4
5  int add(int a, int b)
6  {
7      return a + b;
8  }
9
10 //function which loops many times
11 int f_main_1()
12 {
13     int sum = 0;
14     for(int i = 0; i < MAIN_ITERATIONS; i++){
15         sum += add(1,i);
16     }
17     return sum;
18 }
19
20 //identical to f_main_1 but iterating the other way
21 int f_main_2()
22 {
23     int sum = 0;
24     for(int i = MAIN_ITERATIONS-1; i >= 0; i--){
25         sum += add(1,i);
26     }
27     return sum;
```

```
28 }
29
30 //does half the iterations of f_main
31 int f_half(int start)
32 {
33     int sum = start;
34     for(int i = 0; i < HALF_ITERATIONS; i++){
35         sum += add(1,i);
36     }
37     return sum;
38 }
39
40 //does quarter the iterations of f_main
41 int f_quarter(int start)
42 {
43     int sum = start;
44     for(int i = 0; i < QUARTER_ITERATIONS; i++){
45         sum += add(1,i);
46     }
47     return sum;
48 }
49
50 //runs f_half twice (should be about the same as f_main)
51 int f_main_3()
52 {
53     return f_half(1) + f_half(2);
54 }
55
56 //runs f_quarter 4 times (should be about the same as f_main)
57 int f_main_4()
58 {
59     return f_quarter(1) + f_quarter(2) + f_quarter(3) + f_quarter(4);
```

```
60 }
61
62 //runs f_quarter twice (should be about the same as f_half)
63 int f_half_2()
64 {
65     return f_quarter(1) + f_quarter(2);
66 }
67
68 //sum is there to prevent -O1 optimizations from removing function calls
69 int main(int argc, char **argv)
70 {
71     int sum = 0;
72     //timing of first three should match
73     sum += f_main_1();
74     sum -= f_main_2();
75     sum += f_main_3();
76     sum -= f_main_4();
77
78     //timing should be about half of first ones
79     sum += f_half(0);
80     sum -= f_half_2();
81
82     //timing should be about a quarter of first ones
83     sum -= f_quarter(0);
84
85     return sum;
86 }
```

B.2 WasmProf Validation Output

```
Call Graph:
ind  % time   self      children   called        name
```

```
                                                    <spontaneous>
[0]  100.0   0.000          36663.000  1             _main
             3536.000        3467.000   1/1           _f_main_1 [4]
             3509.000        3479.000   1/1           _f_main_2 [5]
             5223.000        5288.000   1/3           _f_half [3]
             6130.000        6031.000   1/7           _f_quarter [2]
             0.000           6953.000   1/1           _f_main_3 [6]
             0.000           6953.000   1/1           _f_main_4 [7]
             0.000           3477.000   1/1           _f_half_2 [8]
-----------------------------------------------------------------

             3479.048        0.000      10000k/52500k _f_main_1 [4]
             3479.048        0.000      10000k/52500k _f_main_2 [5]
             5218.571        0.000      15000k/52500k _f_half [3]
             6088.333        0.000      17500k/52500k _f_quarter [2]
[1]  49.82   18265.000       0.000      52500k        _add
-----------------------------------------------------------------

             3502.857        3446.286   4/7           _f_main_4 [7]
             1751.429        1723.143   2/7           _f_half_2 [8]
             875.714         861.571    1/7           _main [0]
[2]  33.17   6130.000        6031.000   7             _f_quarter
             45662500000     0.000      17500k/52500k _add [1]
-----------------------------------------------------------------

             3482.000        3525.333   2/3           _f_main_3 [6]
             1741.000        1762.667   1/3           _main [0]
[3]  28.67   5223.000        5288.000   3             _f_half
             91325000000     0.000      15000k/52500k _add [1]
-----------------------------------------------------------------

             3536.000        3467.000   1/1           _main [0]
```

```
[4]  19.10   3536.000     3467.000   1                _f_main_1
             182650000000  0.000      10000k/52500k    _add [1]
-----------------------------------------------------------------

             3509.000     3479.000   1/1              _main [0]
[5]  19.06   3509.000     3479.000   1                _f_main_2
             182650000000  0.000      10000k/52500k    _add [1]
-----------------------------------------------------------------

             0.000        6953.000   1/1              _main [0]
[6]  18.96   0.000        6953.000   1                _f_main_3
             10446.000    10576.000  2/3              _f_half [3]
-----------------------------------------------------------------

             0.000        6953.000   1/1              _main [0]
[7]  18.96   0.000        6953.000   1                _f_main_4
             24520.000    24124.000  4/7              _f_quarter [2]
-----------------------------------------------------------------

             0.000        3477.000   1/1              _main [0]
[8]  9.48    0.000        3477.000   1                _f_half_2
             12260.000    12062.000  2/7              _f_quarter [2]
-----------------------------------------------------------------


Flat Profile:
 %    cumulative   self     called   self       total    name
                                     ms/call    ms/call
49.82   18265.0    18265.0   52500k   0.000      0.000     _add
16.72   12161.0    6130.0    7        875.714    1737.286  _f_quarter
14.25   10511.0    5223.0    3        1741.000   3503.667  _f_half
9.64    7003.0     3536.0    1        3536.000   7003.000  _f_main_1
9.57    6988.0     3509.0    1        3509.000   6988.000  _f_main_2
```

0.00	6953.0	0.000	1	0.000	6953.000	_f_main_3
0.00	6953.0	0.000	1	0.000	6953.000	_f_main_4
0.00	3477.0	0.000	1	0.000	3477.000	_f_half_2
0.00	36663.0	0.000	1	0.000	36663.000	_main

Appendix C

WASABI PROFILER

The following JavaScript code is used in conjunction with a Wasabi instrumented WebAssembly program in order to create a profiler with similar functionality to WasmProf.

```
//check if in node.js
if (typeof window == 'undefined' && typeof module !== 'undefined') {
    var performance = require('perf_hooks')['performance'];
}
//monkey patch the WASM instantiation to add the imports required by the profiler code
let WasmProf = {
    arcs: {}, //actively updated list of arcs
    Arc: class {
        constructor(count, time) {
            this.count = count;
            this.dynamicCount = 0;
            this.time = time;
            this.dynamicTime = 0;
        }
        clone(){
            var copy = new WasmProf.Arc(this.count, this.time);
            copy.dynamicCount = this.dynamicCount;
            copy.dynamicTime = this.dynamicCount;
            return copy;
        }
    },
    fMap: [], //pregenerated function map

    //print function called by user or from wasm code on return if force print was enabled
    printResults: function() {
        var r = WasmProf.saveResults();
        r.callGraph();
        r.flatProfile();
    },

    //result info about a single function
    FunctionResult: class {
        constructor(name) {
            this.name = name;
            this.selfTime = 0.0;
            this.cumulativeTime = 0.0;
            this.called = 0;
            this.children = [];
            this.parents = [];
        }
        clone() {
            var copy = new WasmProf.FunctionResult(this.name);
            copy.selfTime = this.selfTime;
```

```
44              copy.cumulativeTime = this.cumulativeTime;
45              copy.called = this.called;
46              copy.children = this.children.slice();
47              copy.parents = this.parents.slice();
48              return copy;
49          }
50      },
51
52      //result info about all functions
53      Results: class {
54          constructor(funcResultArr, clone) {
55              if (clone) {
56                  this.funcResults = [];
57                  funcResultArr.forEach(function(elem) {
58                      this.funcResult.push(elem.clone())
59                  });
60              } else {
61                  this.funcResults = funcResultArr;
62              }
63
64              //get the 0th element which is the 'external' function
65              this.external = this.funcResults.shift();
66              //remove everything not called
67              this.funcResults = this.funcResults.filter(elem => elem.called > 0);
68
69              //create sorted arrays
70              this.selfTimeSort = this.funcResults.slice();
71              this.selfTimeSort.sort((a,b) => b.selfTime - a.selfTime);
72              this.cumulativeTimeSort = this.funcResults.slice();
73              this.cumulativeTimeSort.sort((a,b) => b.cumulativeTime - a.cumulativeTime);
74          }
75
76          flatProfile(count) {
77              var totTime = (-1 * this.external.selfTime);
78              console.log('Total time: ' + totTime);
79              console.log(' % \tcumulative\tself\tcalled\tself ms/call\t total ms/call\tname');
80              for (var i = 0; i < this.selfTimeSort.length && count != 0; i++) {
81                  if (this.selfTimeSort[i] != undefined && this.selfTimeSort[i].called > 0) {
82                      console.log((100*this.selfTimeSort[i].selfTime/totTime).toFixed(2) + '\t' +
    this.selfTimeSort[i].cumulativeTime.toFixed(3) + '\t\t' +
83                          this.selfTimeSort[i].selfTime.toFixed(3) + '\t' + this.selfTimeSort[i
    ].called + '\t' +
84                          (this.selfTimeSort[i].selfTime/this.selfTimeSort[i].called ).toFixed
    (3) + '\t' +
85                          (this.selfTimeSort[i].cumulativeTime/this.selfTimeSort[i].count).
    toFixed(3) + '\t' + this.selfTimeSort[i].name);
86                      count--;
87                  }
88              }
89          }
90
91          callGraph(count) {
92              function printParent(parent){
93                  if(parent.function.index == undefined) console.log('\t\t\t\t\t\t  <spontaneous>');
94                  else {
```

```
95                      var parentCount = (parent.arc.count + parent.arc.dynamicCount);
96                      var parentTotalTime = (parent.arc.time + parent.arc.dynamicTime);
97                      var parentSelf = parentTotalTime * (curFunc.selfTime/curFunc.cumulativeTime);
98                      var parentChildren = parentTotalTime - parentSelf;
99                      console.log('\t\t' + parentSelf.toFixed(3) + '\t' + parentChildren.toFixed(3)
    + '\t\t' + parentCount + '/' + curFunc.called + '\t' +
100                         parent.function.name + ' ['+parent.function.index+']');
101                 }
102             }
103             function printChild(child){
104                 var childCount = (child.arc.count + child.arc.dynamicCount);
105                 var childTotalTime = (child.arc.time + child.arc.dynamicTime);
106                 var childSelf = childTotalTime * (curFunc.selfTime/curFunc.cumulativeTime);
107                 var childChildren = childTotalTime - childSelf;
108                 console.log('\t\t' + childSelf.toFixed(3) + '\t' + childChildren.toFixed(3) + '\t\
    t' + childCount + '/' + child.function.called + '\t' +
109                         child.function.name + ' ['+child.function.index+']');
110             }
111
112             if(count == undefined) count = 5; //default of 5 entries
113
114             for(var i = 0; i < this.cumulativeTimeSort.length; i++){
115                 if(this.cumulativeTimeSort[i] != undefined && this.cumulativeTimeSort[i].called >
    0) this.cumulativeTimeSort[i].index = i;
116             }
117             var totTime = (-1 * this.external.selfTime);
118             console.log('index\t% time\tself\tchildren\tcalled\tname');
119             for(var i = 0; i < this.cumulativeTimeSort.length && count != 0; i++){
120                 var curFunc = this.cumulativeTimeSort[i];
121                 if(curFunc != undefined && curFunc.called > 0){
122                     curFunc.parents.forEach(printParent);
123                     console.log('['+curFunc.index+']' + '\t' + (100*curFunc.cumulativeTime/totTime
    ).toFixed(2) + '\t' + curFunc.selfTime.toFixed(3) + '\t' +
124                         (curFunc.cumulativeTime - curFunc.selfTime).toFixed(3) + '\t\t' + curFunc.
    called + '\t' + curFunc.name);
125                     curFunc.children.forEach(printChild);
126                     console.log('-----------------------------------------------');
127                     count--;
128                 }
129             }
130         }
131     },
132
133     //saves the current state of the results and return a Results object
134     saveResults: function() {
135         WasmProf.exportData();
136         var functions = [];
137         for (var s in WasmProf.arcs) {
138             if (functions[s] == undefined) {
139                 functions[s] = new WasmProf.FunctionResult(WasmProf.fMap[s]);
140             }
141             for (var d in WasmProf.arcs[s]) {
142                 if (functions[d] == undefined) {
143                     functions[d] = new WasmProf.FunctionResult(WasmProf.fMap[d]);
144                 }
```

```
145                 var count = WasmProf.arcs[s][d].count + WasmProf.arcs[s][d].dynamicCount;
146                 var time = WasmProf.arcs[s][d].time + WasmProf.arcs[s][d].dynamicTime;
147                 if(count > 0){
148                     functions[s].children.push({function: functions[d], arc: WasmProf.arcs[s][d].
    clone()});
149                     functions[s].selfTime -= time;
150 
151                     functions[d].cumulativeTime += time;
152                     functions[d].selfTime += time;
153                     functions[d].called += count;
154                     functions[d].parents.push({function: functions[s], arc: WasmProf.arcs[s][d].
    clone()});
155                 }
156             }
157         }
158 
159         return new WasmProf.Results(functions);
160     }
161 };
162 WasmProf.fMap = [];
163 
164 //make WasmProf accessible in JavaScript console
165 if(typeof window != 'undefined') window.WasmProf = WasmProf;
166 
167 
168 var call_stack = new Array();
169 call_stack.peek = function(){return call_stack[call_stack.length-1];}
170 
171 for(var i = 0; i < Wasabi.module.info.functions.length; i++){
172     var name = i.toString();
173     const fct = Wasabi.module.info.functions[i];
174     if (fct.export[0] !== undefined) name = fct.export[0];
175     else if (fct.import !== null) name = fct.import;
176 
177     WasmProf.fMap[i+2] = name;
178 }
179 WasmProf.fMap[0] = "undefined";
180 WasmProf.fMap[1] = "unknown";
181 WasmProf.exportData = function(){};
182 
183 class call_stack_entry {
184     constructor(id, start_time){
185         this.id = id;
186         this.start_time = start_time;
187     }
188 }
189 call_stack.push(new call_stack_entry(0,0));
190 
191 var forcePrint = false;
192 
193 // happens before each call
194 Wasabi.analysis.call_pre = function (location, func, args) {
195     call_stack.push(new call_stack_entry(func+2, performance.now()));
196 };
197 
```

```
198 //happens after each call
199 Wasabi.analysis.call_post = function(location, value) {
200     var func = call_stack.pop();
201     if(undefined == WasmProf.arcs[call_stack.peek().id]){
202         WasmProf.arcs[call_stack.peek().id] = {};
203     }
204     if(undefined == WasmProf.arcs[call_stack.peek().id][func.id]){
205        WasmProf.arcs[call_stack.peek().id][func.id] = new WasmProf.Arc(0, 0.0);
206     }
207     WasmProf.arcs[call_stack.peek().id][func.id].count += 1;
208     WasmProf.arcs[call_stack.peek().id][func.id].time += (performance.now()-func.start_time);
209 };
210
211 //happens at beginning of each block
212 //this is used to detect initial entry into WASM top-level function
213 Wasabi.analysis.begin = function(location, type) {
214     if(type == "function" && call_stack.length == 1){
215         call_stack.push(new call_stack_entry(1, performance.now()));
216     }
217 }
218
219 //happens right before a function returns
220 //this is used to track when exiting from a top-level WASM function and print results
221 Wasabi.analysis.return_ = function(location, values) {
222     if(call_stack.length == 2){
223         var func = call_stack.pop();
224         if(undefined == WasmProf.arcs[call_stack.peek().id]){
225             WasmProf.arcs[call_stack.peek().id] = {};
226         }
227         if(undefined == WasmProf.arcs[call_stack.peek().id][func.id]){
228         WasmProf.arcs[call_stack.peek().id][func.id] = new WasmProf.Arc(0, 0.0);
229         }
230         WasmProf.arcs[call_stack.peek().id][func.id].count += 1;
231         WasmProf.arcs[call_stack.peek().id][func.id].time += (performance.now()-func.start_time);
232
233         if(forcePrint) WasmProf.printResults();
234     }
235 };
```